**EXPERIMENT NO. 10**

| **Objective(s):**  Implement a Binary Search Tree (BST) data structure to efficiently store and manage elements while preserving the BST properties. |
| --- |
| **Outcome:**  Develop functions for creating a BST, inserting nodes respecting the BST property, and deleting nodes while maintaining the BST structure and properties, ensuring efficient searching, insertion, and deletion operations. |
| **Problem Statement:**  Implement Binary search Tree and its operations ( creation, insertion, deletion). |
| **Background Study:**  **Binary Search Tree (BST)** is a hierarchical data structure that organizes elements in a way that allows for fast search, insertion, and deletion operations. It follows the properties:   1. **Binary Tree Structure:**    * Each node has at most two children: left and right. 2. **Binary Search Property:**    * For every node n, all values in its left subtree are less than n, and all values in its right subtree are greater than n.  Key Operations  1. **Creation:**    * Start with an empty tree or initialize with a root node. 2. **Insertion:**    * Place new elements by comparing with the current node and moving left or right based on the BST properties until an appropriate position is found. 3. **Deletion:**    * Remove nodes while maintaining the BST properties:      + **Case 1:** Node has no children - Remove it directly.      + **Case 2:** Node has one child - Replace it with its child.      + **Case 3:** Node has two children - Find the successor (or predecessor), replace the node with it, and delete the successor (or predecessor). 4. **Traversal:**    * Visit nodes in a specific order:      + **Inorder:** Left subtree, current node, right subtree (sorted order).      + **Preorder:** Current node, left subtree, right subtree (used to create a copy of the tree).      + **Postorder:** Left subtree, right subtree, current node (used in deleting a tree). 5. **Search:**    * Find a specific node based on its value using the BST properties (left for smaller, right for larger).  Balancing  1. **Balanced vs Unbalanced:**    * **Balanced BST:** Ensures that the height of the tree remains logarithmic (O(log n)), optimizing search, insertion, and deletion operations.    * **Unbalanced BST:** May degrade to linear time (O(n)) operations in the worst case, losing the efficiency benefits. 2. **Self-Balancing Trees:**    * **AVL Tree:** Maintains a balance factor for each node to ensure the tree remains balanced after insertions and deletions (O(log n) operations).    * **Red-Black Tree:** Ensures balanced conditions using color attributes on nodes, offering efficient operations (O(log n)).    * **Splay Tree:** Reorganizes itself based on access patterns to bring frequently accessed nodes closer to the root.  Applications  1. **Database Systems:**    * Efficient indexing and searching of records. 2. **Symbol Tables:**    * Storing identifiers in compilers and symbol tables in interpreters. 3. **File Systems:**    * Representing directory structures for efficient file retrieval. 4. **Networking:**    * Routing tables in routers for fast packet forwarding.  Advantages and Disadvantages  * **Advantages:**   + Efficient average-case performance (O(log n) operations).   + Simple implementation of operations. * **Disadvantages:**   + Degradation to O(n) in the worst case (unbalanced tree).   + Extra space for pointers. |

| **Algorithm (Student Work Area):** 1. Creation  * **Algorithm:**   + Start with an empty tree or initialize with a root node.   + If inserting nodes dynamically, allocate memory for each new node and assign values.  2. Insertion  * **Algorithm:**   + Begin at the root.   + Compare the value of the node to be inserted with the current node's value.   + If smaller, move to the left child; if larger or equal, move to the right child.   + Repeat until a suitable empty spot (leaf) is found.   + Insert the new node as a left or right child based on the comparison.  3. Deletion  * **Algorithm:**   + Find the node to delete.   + If the node has no children (leaf node), simply remove it.   + If the node has one child, replace it with its child.   + If the node has two children:     - Find the inorder successor (smallest node in the right subtree) or predecessor (largest node in the left subtree).     - Replace the node to be deleted with the successor or predecessor.     - Recursively delete the successor or predecessor from the subtree.  4. Traversal  * **Inorder Traversal:**   + Visit left subtree.   + Visit the current node.   + Visit right subtree.   + Used to print elements in non-decreasing order. * **Preorder Traversal:**   + Visit the current node.   + Visit left subtree.   + Visit right subtree.   + Used to create a copy of the tree or to prefix expression evaluation. * **Postorder Traversal:**   + Visit left subtree.   + Visit right subtree.   + Visit the current node.   + Used in deleting a tree or for postfix expression evaluation.  5. Searching  * **Algorithm:**   + Start from the root.   + Compare the value of the node to be searched with the current node's value.   + If smaller, move to the left child; if larger, move to the right child.   + Repeat until the node is found or a null pointer is encountered. |
| --- |

| **Code:**  #include <stdio.h>  #include <stdlib.h>  // Definition of a BST node  struct TreeNode {  int data;  struct TreeNode \*left;  struct TreeNode \*right;  };  // Function to create a new node  struct TreeNode\* createNode(int data) {  struct TreeNode\* newNode = (struct TreeNode\*) malloc(sizeof(struct TreeNode));  newNode->data = data;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  // Function to insert a new node in BST  struct TreeNode\* insertNode(struct TreeNode\* root, int data) {  // If the tree is empty, return a new node  if (root == NULL) {  return createNode(data);  }  // Otherwise, recur down the tree  if (data < root->data) {  root->left = insertNode(root->left, data);  } else if (data > root->data) {  root->right = insertNode(root->right, data);  }  // return the (unchanged) node pointer  return root;  }  // Function to find the inorder successor in BST  struct TreeNode\* minValueNode(struct TreeNode\* node) {  struct TreeNode\* current = node;  // Loop down to find the leftmost leaf  while (current && current->left != NULL) {  current = current->left;  }  return current;  }  // Function to delete a node from BST  struct TreeNode\* deleteNode(struct TreeNode\* root, int data) {  // Base case: If the tree is empty  if (root == NULL) {  return root;  }  // Recur down the tree  if (data < root->data) {  root->left = deleteNode(root->left, data);  } else if (data > root->data) {  root->right = deleteNode(root->right, data);  } else {  // Node found with the data  // Case 1: Node with only one child or no child  if (root->left == NULL) {  struct TreeNode\* temp = root->right;  free(root);  return temp;  } else if (root->right == NULL) {  struct TreeNode\* temp = root->left;  free(root);  return temp;  }  // Case 2: Node with two children  struct TreeNode\* temp = minValueNode(root->right);  // Copy the inorder successor's content to this node  root->data = temp->data;  // Delete the inorder successor  root->right = deleteNode(root->right, temp->data);  }  return root;  }  // Function to perform inorder traversal of BST  void inorderTraversal(struct TreeNode\* root) {  if (root != NULL) {  inorderTraversal(root->left);  printf("%d ", root->data);  inorderTraversal(root->right);  }  }  // Function to search for a node in BST  struct TreeNode\* search(struct TreeNode\* root, int data) {  // Base Cases: root is null or data is present at root  if (root == NULL || root->data == data) {  return root;  }  // data is greater than root's data  if (root->data < data) {  return search(root->right, data);  }  // data is smaller than root's data  return search(root->left, data);  }  // Example usage  int main() {  struct TreeNode\* root = NULL;  root = insertNode(root, 50);  insertNode(root, 30);  insertNode(root, 20);  insertNode(root, 40);  insertNode(root, 70);  insertNode(root, 60);  insertNode(root, 80);  printf("Inorder traversal of the BST: ");  inorderTraversal(root);  printf("\\n");  // Search for a node  int key = 40;  struct TreeNode\* result = search(root, key);  if (result) {  printf("Node %d found in the BST.\\n", key);  } else {  printf("Node %d not found in the BST.\\n", key);  }  // Delete a node  root = deleteNode(root, 20);  printf("Inorder traversal after deleting 20: ");  inorderTraversal(root);  printf("\\n");  return 0;  } |
| --- |
| **OUTPUT :** |